VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Hledání podmnožiny databáze s ohledem na vytížení

Database Subsetting with Respect to the Workload

2018 Dominik Kaluža

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Dominik Kaluža**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Hledání podmnožiny databáze s ohledem na vytížení
Database Subsetting with Respect to the Workload

Jazyk vypracování: čeština

Zásady pro vypracování:

Pro hledání podmnožiny databáze (tzv. database subsetting) existuje celá řada nástrojů. Cílem této práce je podrobněji prozkoumat tři database subsetting nástroje a navrhnout řešení, které umožní vytvářet podmnožiny databáze s ohledem na existující vytížení.
Hlavním požadavkem na výslednou podmnožinu databáze je:
1. SQL příkazy z vytížení, které vracejí neprázdné výsledky na celé databázi, budou vracet neprázdné výsledky i na její podmnožině.
2. Každý SQL příkaz z vytížení by měl mít stejný plány vykonání na obou databázích.

Při řešení se bude postupovat v následujících krocích>
1. Rešerše nástrojů pro database subsetting.
2. Formulování požadavků na databse subsetting nástroj s ohledem na vytížení.
3. Analýza, návrh a implementace nástroje, který provede database subsetting s ohledem na vytížení.
4. Důkladné otestování nástroje.

Seznam doporučené odborné literatury:

[1] Subsetting v nástroji Toad for Oracle: http://www.toadworld.com/products/toad-for-oracle/b/weblog/archive/2014/03/20/toad-can-copy-a-subset-of-production-data-to-development-and-test-databases

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Radim Bača, Ph.D.**

Datum zadání: 01.09.2016
Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2018 .....

Rád bych na tomto místě poděkoval Ing. Radimovi Bačovi, Ph.D. za odborné vedení a za pomoc a rady při zpracování této práce.

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací nástroje pro hledání podmnožiny databáze s ohledem na vytížení. První část práce se zabývá problematikou subsettingu a rešerší existujících nástrojů s ohledem na naše požadavky. V druhé části je popis samotné implementace nástroje a řešení problémů, které nastaly. Součástí je také otestování nástroje a analýza získaných dat.

Klíčová slova: databáze, podmnožina, vytížení, SQL, SQL Server, subsetting

Abstract

This thesis describes the design and implementation of a tool for database subsetting with respect to workload. The first part deals with problematics of subsetting and the analysis of existing tools with regard to our requirements. In the second part is a description of the implementation itself and the solution to problems that have occurred. It also includes testing of tools and analysis of acquired data.

Key Words: database, subset, workload, SQL, SQL Server, subsetting

Obsah

Seznam použitých zkratek a symbolů

ADO.NET	–	Microsoft ActiveX Data Objects .NET
ANTLR	–	ANother Tool for Language Recognition
CRUD	–	Create, Read, Update, Delete
CSV	–	Comma-separated values
DBMS	–	Database Management System
DBRG	–	Database Research Group
DLL	–	Data Definition Language
ET	–	Execution Time
GUI	–	Graphical User Interface
ID	–	Identity
IEP	–	Identical Execution Plan
IRC	–	Identical Result Count
JDBC	–	Java Database Connectivity
LINQ	–	Language Integrated Query
MSSQL	–	Microsoft SQL Server
MVVM	–	Model-view-viewmodel
PDF	–	Portable Document Format
SMO	–	SQL Server Management Objects
SQL	–	Structured Query Language
UI	–	User Interface
UUID	–	Universally unique identifier
T-SQL	–	Transact-SQL
WPF	–	Windows Presentation Foundation
XAML	–	Extensible Application Markup Language
XML	–	Extensible Markup Language

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Důsledkem zvyšujících se požadavků byznysu, databáze i aplikace, které data zpracovávají, neustále rostou v jak velikosti, tak složitosti. S rostoucí složitostí se stává důkladné testování důležitým aspektem kvality softwaru. Obecnou praxí je využívání, v lepších případech, kopie produkční databáze. Odůvodněno to bývá tím, že testujeme na reálných a aktuálních datech.

Pokud ovšem databáze překročí určitou mez ve velikosti, stává se zajišťování plnohodnotných databází obtížnou úlohou například z důvodu nedostatku úložného prostoru nebo času. Jednoduchým řešením tohoto problému je mít malý počet těchto databází a sdílet je mezi vývojáři a testery. Nevýhodou tohoto řešení je, že data se brzy stanou neaktuálními.

Testovat na plnohodnotných databázích je málokdy potřebné, většinou si vystačíme s kopií, obsahující data k otestování konkrétního problému nebo situace. Není ovšem jednoduché exportovat malou část databáze. Kdybychom náhodně vybrali 10% záznamů z každé tabulky, data by nesplňovala referenční integritu.

Cílem této práce je navrhnout a vytvořit nástroj pro automatické zkopírování podmnožiny dat s ohledem na vytížení. Vytížení je v našem případě seznam SQL příkazů. To znamená, že příkazy vrací na obou databázích nejlépe stejný počet výsledků. Druhý požadavek je, že každý příkaz má identický plán vykonání.

První kapitola je zaměřena na problematiku database subsettingu, jeho výhody, nevýhody a problémy, které mohou nastat. V druhé kapitole je rešerše již existujících nástrojů, jejich porovnání a zhodnocení vzhledem k našim požadavkům. V kapitole 3 jsou krátce popsány a vysvětleny požadavky na výsledný nástroj.

Ve čtvrté kapitole jsou představeny použité technologie a samotná implementace je popsána v páté kapitole. Důraz je kladen na výběr vhodného parseru *SQL* a důkladně je popsána architektura programu s podrobnějším popisem složitějších funkcí programu. Dále jsou zde popsány problémy, které při implementaci a testování nastaly a jejich řešení. Šestá kapitola obsahuje stručnou ukázku výsledné aplikace. V poslední kapitole se nachází výstupy testování nástroje a jejich analýza.

2 Database Subsetting

Podkapitoly 2.1 a 2.2 jsou založeny na informacích z článku *Data Subsetting* [1].

2.1 Kdy provést?

Existuje množství důvodů, proč bychom chtěli provést database subsetting (dále jen *subsetting*). Nejčastější důvody jsou popsány níže.

2.1.1 Nedostatek úložného prostoru pro kopii databáze

Toto je důvod, který napadne skoro každého. Databáze jednoduše zabírá více místa na disku, než je k dispozici v prostředí, do kterého chceme databázi přenést. V dnešní době se objem dat i počet aplikací neustále zvyšuje a pokud nemáme potřebný úložný prostor, neexistuje žádná jiná možnost, než provést subsetting.

2.1.2 Nedostatek času

I v případě, že máme k dispozici dostatek místa, není duplikování produkční databáze vždy ta pravá volba. Kopírování databáze o velikosti několik terabajtů může trvat zbytečně dlouho ve srovnání se zkopírováním podmnožiny. Tento problém se může znásobit tím, pokud každý vývojář potřebuje mít aktuální databázi ve svém prostředí. Bez subsettingu může tento proces spotřebovat hodně času databázového administrátora, který by jen kopíroval a zajišťoval, že vše funguje.

2.1.3 Tvorba testovacího prostředí

Pokud pravidelně tvoříme funkční zkušební prostředí pro testování nových verzí aplikací, může subsetting ušetřit spoustu času a místa. V tomto případě je dobré data navíc zamaskovat pro minimalizaci šance úniku citlivých informací.

2.1.4 Vytváření vzdělávacího prostředí

Při školení zaměstnanců o používání aplikace je dobré pokud máme databázi s realistickými daty. Obvykle však v tomto prostředí není potřeba mít celou produkční databázi a vystačíme si s podmnožinou. Zde se podle okolností také mohou data maskovat.

2.1.5 Důkazy konceptů (*Proof of concept*)

Ať už testujeme nový nápad na stávající aplikaci, nebo se snažíme zjistit, jestli aplikace nového dodavatele bude vyhovovat našim potřebám, je nejlepší možnost provést test s daty, která jsou kritická pro správné fungování systému nebo aplikace. Pro vybrání těchto dat je database subsetting ideální volba.

2.1.6 Opakované přenosy identických dat

Mohou nastat různé situace, kdy několikrát klonujeme stejná data:

- Máme více vývojářů a každý z nich pracuje na jiné části aplikace. Chceme, aby každý měl svou vlastní množinu dat, kterou si případně podle potřeby aktualizuje.
- Při využití agilní metodiky vývoje chceme rychle testovat nové (*přírůstkové*) změny. Pokud se tenhle případ opakuje dost často, může se stát, že nemáme dost času na klonování produkční databáze. Pokud chceme opakovaně klonovat stejnou množinu dat, měli bychom zvážit použití database subsettingu.

2.2 Proč nesubsettovat?

Může se zdát, že subsetting je užitečný v každé situaci, ale není to pravda. Níže jsou dva případy, kdy chceme využít celou produkční databázi.

2.2.1 Testování zátěže a výkonu

Nejpřesnější způsob jak otestovat výkon a běh aplikace pod zatížením, je mít databázi s plnohodnotnými a realistickými daty. Tyto dvě podmínky musí být vždy dodrženy.

Často se stává, že při vývoji aplikace běží perfektně, protože je na vývojovém prostředí. Jakmile aplikaci spustíme v produkčním prostředí zpozorujeme zpomalení. Důvodem tohoto chování je, že vývojové databáze jsou obecně mnohem menší, takže nám optimalizační problémy mohou uniknout. Data mohou být tak malá, že můžeme přehlédnout například chybějící indexy, protože *table scan* funguje dostatečně dobře při malém počtu záznamů.

Nestačí však mít pouze dostatečně velkou databázi. Existuje mnoho nástrojů pro generování dat různých velikostí, ale tato data nejsou realistická. Data by měla být stejná, případně alespoň podobná. Pokud tomu tak není, stejný dotaz může mít kompletně rozdílný plán vykonání a takové chování je při testování výkonu nežádoucí.

Pokud data navíc maskujeme, musíme zachovat základní vlastnosti těchto dat. Nemůžeme například měnit 9místné čísla na 2místné.

2.2.2 Krajní případy (*Edge Cases*)

Druhý scénář, kde je výhodou mít celou produkční databázi je, když potřebujeme otestovat krajní případy (*edge cases*). Pro představu takové situace je níže jednoduchý příklad.

```
1 pole ← 10prvkové pole čísel
2 for i ← 1 to 10 do
3 |   pole[i] ← -1
4 end
```

Algoritmus 1: Naplnění 10prvkového pole

Myšlenka kódu 1 je vytvořit 10prvkové pole a inicializovat každý prvek na -1. Problém je, že kód neinicializuje první prvek (`pole[0]`), protože smyčka začíná indexem 1.

Při pohledu na toto malé pole lze vidět, že krajní případ je méně než 10% dat. V realističtějších datech krajní případy reprezentují mnohem menší podíl a provedení subsettingu na takovýchto datech může zredukovat šanci, kdy na tento případ narazíme nebo šanci úplně eliminovat. Tuto chybu bychom při testování nejspíše neodhalili a to je samozřejmě nežádoucí.

2.3 Problémy při subsettingu

Existuje řada problémů, které je při database subsettingu nutno zvážit. Následující podkapitoly jsou založeny na informacích z práce *Database Subsetting: What You Need to Know* [2].

2.3.1 Duplicitní záznamy

Je poměrně běžné, že tabulka má dva nebo více rodičů. To znamená, že potomek musí obsahovat všechny záznamy, které jsou referencovány v rodičovských tabulkách. Při procesu tvorby podmnožiny je potomek procházen pro každého rodiče zvlášť a proto může nastat situace, kdy se přenese záznam, který již v potomkovi existuje. Problém je vysvětlen na jednoduchém příkladu níže.

Tabulky 1, 2 a 3 reprezentují databázové tabulky *objednavka*, *zakaznik* a *zamestnanec*, kde objednávka má relaci *n:1* na zákazníka a *n:1* na zaměstnance. Pokud bude v podmínkách subsettingu, aby v nové databázi byl zákazník s id 1 a jeho objednávky a zároveň zaměstnanec s id 2 a objednávky, které odbavuje, přenese se nejdříve zákazník s id 1 a objednávka s id 1. Poté se přenese zaměstnanec s id 2 a objednávky s id 1 a 2. V tomto případě bude duplicitní objednávka s id 1.

Tabulka 1: Tabulka zakaznik

zakaznik	
id	*jmeno*
1	Pepa Novák
2	Josef Novotný

Tabulka 2: Tabulka zamestnanec

zamestnanec	
id	*jmeno*
1	David Nguyen
2	Lukáš Malý

Tabulka 3: Tabulka objednavka

objednavka			
id	*produkt*	*id_zakaznik*	*id_zamestnanec*
1	mixér	1	2
2	odšťavňovač	2	2

Duplicitní záznamy v tabulce být nemohou, přinejmenším by zabránily aktivování omezení primárních a cizích klíčů.

Existuje více možností, jak tento problém řešit. Prvním způsobem je odstranění duplicitních záznamů po jejich vložení do databáze. Nevýhodou je zbytečný nárůst časové náročnosti subsettingu — nejprve se záznamy vloží a poté se zase smažou. Tou lepší možností je vyfiltrování duplicit. Tohoto lze docílit tím, že duplicitní záznamy před přenosem vyřadíme nebo si je vůbec nevyžádáme.

2.3.2 Cyklus v databázi

Schéma databází jsou většinou velmi komplexní. Často se v nich vyskytují cykly, kdy například tabulka A odkazuje na tabulku B, tabulka B na tabulku C a tabulka C na tabulku A. Cykly mohou samozřejmě obsahovat daleko větší množství tabulek.

Důsledkem tohoto zacyklení může být i kompletní přenos zdrojové databáze do subsetu. K tomu dojde tak, že se neustále kopírují data potřebná pro zachování referenční integrity. Takový subsetting je naprosto zbytečný.

2.3.3 Rekurzivní relace

Rekurzivní relace je v podstatě cyklus, ale tento případ může být daleko horší pro běh programu. Při každé iteraci buď hledáme nadřazené, nebo podřazené záznamy pro aktuální záznam a při každé další iteraci se tento proces opakuje. Důvodem proč je tahle situace horší než prostý cyklus, je malý počet kopírovaných záznamů při každé iteraci. Iterací pak může být velmi velký počet a to značně ovlivní časovou náročnost subsettingu.

3 Rešerše

3.1 Jailer

Jailer je open-source program napsaný v Javě pro hledání podmnožin databáze. Umí exportovat konzistentní podmnožiny s neporušenou referenční integritou. Výhodami nástroje je multiplatformnost a schopnost pracovat s širokou škálou dnes využívaných DBMS k čemuž využívá ovladače *JDBC*.

Konfigurace subsettingu je uložena v *extraction modelu*, což je *CSV* soubor obsahující seznam tabulek a podmínek k jednotlivým tabulkám. Dále jsou zde vydefinované vazby ve formě cizích klíčů a vygenerované inverzní vazby tzn. v případě, že máme v databázi cizí klíč Aukce → Vlastník, vytvoří se i vazba Vlastník → Aukce. Ve výchozím nastavení program kopíruje všechna relevantní data a právě tomuto můžeme zabránit ignorováním určitých vazeb.

Při použití nástroje může nastat situace, kdy nenalezne všechny tabulky a relace, ty je pak nutno přidat manuálně. Další nevýhodou je, že program nepodporuje přenos dat do jiné databáze, pouze do jiného schéma stejné databáze. Další možností je export *DLL* scriptu s požadovanými daty. V obou případech si musíme předpřipravit strukturu cílové databáze.

Zásadní nevýhodou s ohledem na naše potřeby je nemožnost vložit vytížení ve formě SQL dotazů. Dotazy by bylo nutné rozparsovat a spojit v podmínky na jednotlivé tabulky a v případě příkazů, které spojují více tabulek, správně nastavit vazby a to vše vložit do *extraction modelu*. Dále by bylo nutno vytvořit příkaz pro spuštění nástroje a ten vykonat v příkazové řádce.

3.2 Toad for Oracle

Toad for Oracle je rozsáhlý nástroj užívaný vývojáři, databázovými administrátory a analytiky dat pro práci s relačními i nerelačními databázemi. Program obsahuje spoustu funkcí pro jednodušší design a vývoj, automatizovanou analýzu a testování SQL kódu, identifikování limitů tzv. *bottlenecků* v databázi a další.

Jednou z funkcí je *Data Subset Wizard*. Utilita nabízí výrazně omezenější nastavení v porovnání s ostatními testovanými nástroji. Pro konfiguraci subsettingu slouží jen dvě pole — procento vybraných dat a minimální počet záznamů. Z těchto dvou hodnot se použije ta, která je na dané tabulce vrací větší množství záznamů.

Možnostem nastavení odpovídá i výsledek. Ve většině případů je přeneseno značně více dat, než je potřeba. Když nastavíme, že chceme např. 10% záznamů, program přenese 10% záznamů z každé tabulky, ale poté přenese ještě relevantní data tak, aby zůstala neporušena referenční integrita.

Tímto nástrojem nelze provést database subsetting s ohledem na vytížení, pouze s ohledem na velikost databáze. Proto naprosto nevyhovuje našim potřebám.

3.3 DataBee

DataBee je komerční database subsetting software od firmy Net2000. Podpora zahrnuje DBMS SQL Server a Oracle. Pro rešerši byla využita 30denní trial verze, kterou může využít každý, kdo zašle žádost na jejich oddělení podpory.

Po instalaci máme na výběr ze tří podprogramů — *Set Designer*, *Set Extractor* a *Set Loader*. Nás budou zajímat nástroje *Set Designer*, který slouží pro navrhnutí pravidel pro subsetting a *Set Extractor*, který provede subsetting na základě těchto nadefinovaných pravidel.

Nástroj je celkově velice rozsáhlý a složitý a jeho nastavení není lehká záležitost. Sami vývojáři nabízí na svých stránkách bezplatnou pomoc po emailu.

S ohledem na naše potřeby využijeme opravdu malý zlomek možností, které *DataBee* nabízí. Program funguje na podobném principu jako Jailer (viz kapitola 3.1). Lze vytvořit pravidla pro jednotlivé tabulky a to buď klauzulí where, zadáním požadovaného procenta záznamů nebo počtem řádků. Toto nastavení se poté uloží do *extraction setu*, což je rozsáhlý XML soubor, který obsahuje veškeré cizí klíče, tabulky, pravidla pro tyto tabulky a další. Posledním krokem je spustit *Set Extractor*, načíst tento soubor a spustit database subsetting.

Problém s tímto typem nastavení je stejný jako u *Jaileru* — nemožnost vytvořit podmínky z vytížení. Vytížení by se muselo opět rozparsovat a vložit do *extraction setu*, a protože program nenabízí spuštění z příkazové řádky, bylo by ho nutné spustit manuálně. Nástroj také nepodporuje subsetting do jiné databáze, pouze do jiného schéma.

3.4 Shrnutí

Po analýze těchto tří nástrojů lze říct, že ani jeden nesplňuje všechny podmínky této práce. Z části by se daly využít nástroje *Jailer* a *DataBee*, kde je možné nastavit podmínky pro jednotlivé tabulky a v případě jednoduchých dotazů ve vytížení by subsetting proběhl bez problému. *Toad for Oracle* nevyhovuje požadavkům ani z části.

Problém, který by bylo nutno řešit je naparsování jednotlivých příkazů a pospojování jejich where klauzulí, tak aby vznikla vždy jedna podmínka pro každou tabulku v databázi.

Druhý, složitější problém je vložit tyto podmínky do *extraction modelu* nebo *extraction setu*. Formáty těchto souborů jsou komplikované, protože se počítá s tím, že uživatel všechno nastavení provede přes uživatelské rozhraní. Dodat data do těchto souborů programově by proto mohl být velice náročný úkol.

Na základě výše uvedených nedostatků vyplývá závěr, že nástroje jsou nedostatečné pro řešení problému této práce, mnohdy zbytečně komplexní, a že nejlepší řešení je naimplementovat nástroj na míru. Shrnutí rešerše nástrojů je v tabulce 4.

Tabulka 4: Porovnání existujících nástrojů

Funkce	**Jailer**	**Toad for Oracle**	**DataBee**
Subsetting do jiné databáze	Ne	Ne	Ne
Subsetting do jiného schéma stejné databáze	Ano	Ano	Ano
Podmínky pro jednotlivé tabulky	Ano	Ne	Ano
Zachovaná referenční integrita	Podle nastavení	Ano	Podle nastavení
Počet podporovaných DBMS	10	1	2
Složitost nastavení v uživatelském rozhraní	Složité	Jednoduché	Průměrné
Možnost spuštění v příkazové řádce	Ano	Ne	Ne
Odhadovaná složitost programového nastavení	Složité	Nemožné	Velice složité
Nutnost předpřipravit schéma databáze	Ano	Ne	Ne
Subsetting s ohledem na vytížení	Ne	Ne	Ne

4 Požadavky

Hlavními požadavky na výslednou podmnožinu databáze je:

- Každý SQL dotaz z vytížení, který vrací neprázdné výsledky na zdrojové databázi, musí vracet neprázdné výsledky na výsledné podmnožině databáze
- Každý SQL příkaz by měl mít stejný plán vykonání na zdrojové databázi i podmnožině

4.1 Vytížení

Vytížení v našem případě znamená seznam SQL dotazů v souboru typu SQL nebo XML. Pro programové zpracování obou typů vytížení jsou definovány určité podmínky.

4.1.1 SQL formát

Tento formát má jedinou podmínku: příkazy musí být odděleny řetězcem `--delimiter--`. Příklad jednoduchého vytížení můžeme vidět ve výpisu 1.

```
SELECT * FROM PizzaSurovina ps
JOIN Pizza p ON p.ID_Pizza = ps.ID_Pizza
WHERE ID_Surovina > 5 AND ID_Surovina < 8
AND Velikost > 40;
--delimiter--
SELECT * FROM PizzaSurovina
WHERE ID_Surovina > 12 AND ID_Surovina < 14;
--delimiter--
SELECT * FROM Zakaznik WHERE Jmeno LIKE 'Do%';
```

Výpis 1: Příklad SQL souboru s vytížením

4.1.2 XML formát

Tento formát má specifickou strukturu, díky které je jednoduché definovat velké množství dotazů. Uzly a atributy potřebné ke správné konfiguraci jsou:

- Kořenem dokumentu je uzel `Workload`.
- Uzel `sql` obsahuje atribut `text`, ve kterém je předepsán SQL dotaz. Hodnoty, které chceme dynamicky měnit nahradíme znakem `?`.
- Do uzlu `sql` dále zanořujeme uzel `values`.

- Uzel `values` obsahuje stejné množství potomků — uzlů `p` — jako počet `?` v dotazu. Při programovém zpracování souboru se hodnoty potomků postupně doplní za výskyty `?` v dotazu.

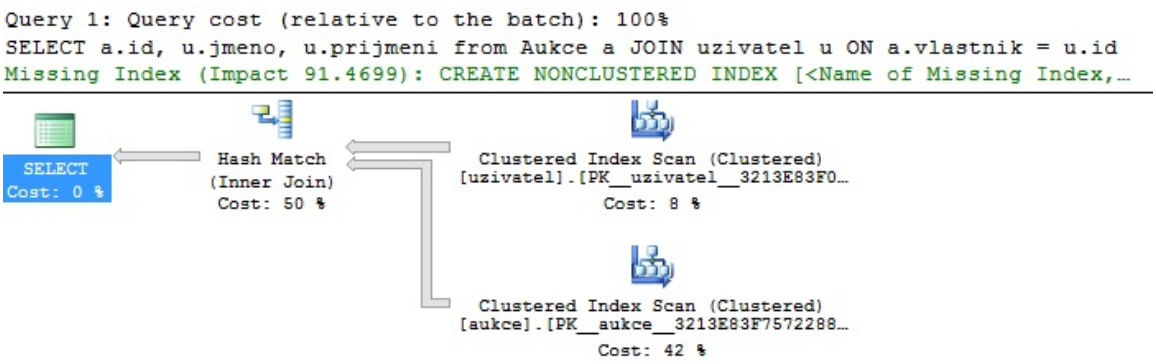
Tento typ vytížení je používán skupinou *Database Research Group (DBRG)*, ale pro potřeby nástroje byl značně zjednodušen. Příklad můžeme vidět na výpisu 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<Workload>
   <sql text="SELECT * FROM uzivatel WHERE id = ?">
      <values>
         <p>825</p>
      </values>
      <values>
         <p>52</p>
      </values>
    </sql>
    <sql text="SELECT * FROM prihoz WHERE aukceid = ? OR uzivatelid = ?">
        <values>
            <p>10</p>
            <p>825</p>
        </values>
        <values>
            <p>84</p>
            <p>877</p>
        </values>
    </sql>
    <sql text="SELECT * FROM kategorie" />
</Workload>
```

Výpis 2: Příklad XML souboru s vytížením

4.2 Plán vykonání

Než se každý dotaz provede, projde nejprve optimalizátorem dotazů (*query optimizer*). Ten určí logiku provedení dotazu, tzn. určí logické operace a kroky. Výsledkem této operace je plán vykonání. Je to uspořádaná množina kroků, které DBMS provede aby vykonal SQL dotaz. Tento proces není perfektní, a proto je někdy nutné plán prozkoumat, zjistit problém a optimalizovat dotaz manuálně. Taky z něj lze zjistit chybějící indexy, např. výpisy plánu vykonání na MSSQL databázi obsahují doporučení indexů. Existuje několik typů: textový, XML a grafický. Na obrázku 1 je příklad grafického plánu vytížení.



Obrázek 1: Příkad grafického zobrazení plánu vykonání

5 Využité technologie a nástroje

Pro implementaci desktopové aplikace byla zvolena technologie WPF, což je knihovna pro tvorbu uživatelského rozhraní, která je součástí .NET frameworku. Ve WPF se využívá návrhový vzor MVVM a technika *Data Binding*, která zajišťuje obousměrnou synchronizaci dat a uživatelského rozhraní. Druhou možností tvorby uživatelského rozhraní je knihovna Windows Forms (WinForms), která je však v dnešní době zastaralá. V aplikaci jsou dále využity knihovny *SQL Server Management Objects* a *General SQL Parser* a další. Při vývoji aplikace byl využit verzovací systém GIT.

5.1 SQL Server Management Objects (SMO)

SQL Server Management Objects (SMO) z dílny Microsoftu je knihovna objektů vyžívaných při vývoji aplikací pracujících s SQL Server databází. Knihovna obsahuje dva typy tříd - instanční a pomocné.

Instanční třídy reprezentují objekty MSSQL databáze jako servery, databáze, tabulky, triggery a uložené procedury. Tyto třídy jsou hierarchicky uspořádány stejně jako databázový server.

Pomocné třídy byly vytvořeny výslovně pro vykonávání specifických úkolů. Zde patří:

- `Transfer` — používá se pro přenos schéma a dat do jiné databáze.
- `Backup a Restore` — používané pro zálohu a obnovení databáze.
- `Scripter` — pro práci a tvorbu DLL script souborů.

Alternativou k tomuto přístupu je si objekty (modely) vytvořit v aplikaci sám a data do nich načítat ze systémového katalogu databáze nebo využít knihovnu *ADO.NET*. Jeden z těchto přístupů by bylo nutno implementovat v případě rozšíření podpory pro více DBMS.

Knihovnu SMO jsem zvolil proto, že jsem implementoval podporu jen pro SQL Server a v tomto případě značně ulehčuje komunikaci a práci s databází.

5.2 Parsery

Parsování, neboli syntaktická analýza, označuje v informatice a lingvistice proces, kterým analyzujeme posloupnost symbolů v přirozeném jazyce, počítačovém jazyce nebo datových strukturách, které dodržují pravidla formální gramatiky. Výstupem syntaktické analýzy v oboru informatiky je derivační strom, abstraktní syntaktický strom nebo jiná hierarchická struktura. Z této struktury jsme pak schopni zjistit význam vstupu. Pro parsování *SQL* v jazyce *C#* existuje několik nástrojů. Při analýze a výběru byl hlavní faktor dokumentace.

5.2.1 Třída TSqlParser

Třída `TSqlParser` je obsažena v knihovně `Microsoft.SqlServer.TransactSql.ScriptDom`. Velkou nevýhodou je nedostatečná dokumentace. Oficiální dokumentace od Microsoftu popisuje pouze metody této třídy, ale nezachází do hloubky. Jakákoliv implementace by tedy byla založena na metodě pokus-omyl. Skutečnost, že třídou lze analyzovat pouze gramatiku *T-SQL* není v našem případě nevýhodou.

5.2.2 ANTLR (ANother Tool for Language Recognition)

ANTLR je generátor parserů a lexerů strukturovaného textu nebo binárních souborů. Je hojně používaný v akademické i průmyslové sféře. Například vyhledávač na Twitteru pomocí *ANTLRu* parsuje přes 2 miliardy dotazů denně. [3] Skládá se ze dvou částí: nástroje, který přeloží danou gramatiku na parser nebo lexer a knihovny, která je potřebná pro funkci těchto vygenerovaných parserů a lexerů. [4]

S dokumentací je na tom lépe než *Třída TSqlParser*. Dokumentace je ovšem zaměřena na nástroj samotný a na jeho možnosti. Dozvědět se z ní dá jak s *ANTLRem* začít, jak funguje, jaké má nastavení, co je výsledkem analýzy. Mnohem rozsáhlejší dokumentaci obsahuje kniha *The Definitive ANTLR 4 Reference* od Terrence Parra, tvůrce *ANTLRu*. K jejímu využití je ale nutno si knihu koupit. Samotný *ANTLR* je nástroj pro generování parserů a lexerů z gramatik, tudíž neposkytuje dokumentaci k vygenerovanému kódu, tu by měli zajistit tvůrci gramatiky. Většina gramatik jsou komunitní open-source projekty a nemusí být kompletní. Toto platí i v případě T-SQL gramatiky, kde důvodem je nekompletní reference gramatiky ze strany Microsoftu. [5] Tvůrci této gramatiky neposkytují žádnou dokumentaci, takže implementace by byla opět metodou pokus-omyl.

5.2.3 General SQL Parser

General SQL Parser je knihovna pro parsovaní SQL vyvíjena firmou Gudu Software. Je to komerční software s cenou za licenci od 500$ na rok.

Tento nástroj vyšel z analýzy s nejlepšími výsledky. Na webových stránkách nástroje lze nalézt základní dokumentaci potřebnou k instalaci, popis funkčnosti a popis práce s parserem. V tomto případě je tato dokumentace přínosnější než u *ANTLRu*, protože nástroj se zaměřuje pouze na analýzu SQL gramatiky. Největší výhodou je velké množství praktických příkladů na webových stránkách výrobce. Příklady jsou ve většině případů napsány v Javě, ale nástroj má podobné rozhraní i v ostatních jazycích, takže to není problém. Příklady popisují analýzu CRUD dotazů a dotazů s množinovým operátorem, získání atributů v select listu, nahrazení těchto atributů, modifikování where klauzule a další použití. Tyto dema lze i stáhnout a vyzkoušet.

V aplikaci je využita trial verze, která postačuje našim potřebám.

5.3 Material Design In XAML Toolkit

Material Design In XAML Toolkit je jedna z nejpopulárnějších GUI knihoven pro WPF. Mezi výhody patří, že je open-source, lehce použitelná a k dispozici existuje projekt, kde jsou použity všechny možnosti této knihovny.

5.4 StyleCop

StyleCop je open-source nástroj pro statickou analýzu kódu, který kontroluje předem nastavené standardy a zajišťuje konzistenci zdrojového kódu nezávisle na autorovi. Nástroj vynucuje konvence pojmenovávání proměnných a funkcí, seřazení metod ve třídách, čitelnost kódu a další. Dodržování těchto pravidel je základem každého kvalitně napsaného projektu.

6 Implementace

6.1 Architektura

Celý solution je rozdělen do dvou projektů — *Core* a *DesktopApp*, které jsou popsány v kapitolách 6.2 a 7.

Při raném stádiu vývoje existoval ještě třetí projekt *ConsoleApp*, který sloužil k testování připojení, klonování databáze a parsovaní dotazů.

V implementaci je použita řada návrhových vzorů a při návrhu bylo myšleno na to, aby v případě potřeby, byl nástroj jednoduše rozšiřitelný např. podporou dalších typů dotazů.

6.2 Knihovna Core

Core je jádrem celého nástroje a poskytuje programové rozhraní, v našem případě pro projekt *DesktopApp*. Výhodou je, že v případě potřeby lze celé uživatelské rozhraní přepsat bez nutnosti zasahovat do jádra.

Třídy jsou rozděleny do odpovídajících jmenných prostorů (namespace), které mají prefix `DatabaseSubsettingTool.Core`. V dalším popisu architektury v této kapitole bude prefix vynechán. Níže si rozebereme funkce jednotlivých jmenných prostorů a tříd a problémů, které bylo nutné řešit.

6.2.1 Enum

Výčtové typy jsou v tomto projektu použity především pro zlepšení čitelnosti kódu.

6.2.1.1 Enum.Database.AuthMode

Slouží pro rozeznání typu připojení k databázi — Windows Authentication a SQL Server Login.

6.2.1.2 Enum.Database.IgnoreDuplicateKeys, Enum.Database.ReferentialIntegrityMode

Mají hodnoty Enable a Disable a používají se v případech, kdy jedna metoda může něco zapnout i vypnout.

6.2.1.3 Enum.Workload.SqlQueryErrorMessageType

Určuje typ chyb, které mohou nastat při zpracování vytížení a samotném subsettingu. Konkrétní použití je dále vysvětleno v kapitole 6.2.6.3.

6.2.2 Exception

V průběhu aplikace mohou nastat situace kdy program vyhodí výjimku. Tyto výjimky jsou vyhazovány v případě, kdy není splněna nějaká podmínka pro správné provedení subsettingu, např. se nepodařilo správně zpracovat soubor s vytížením, dotaz neprojde validací nebo při nesprávných výstupech při testování po subsettingu. Všechny výjimky, které mohou nastat při zpracování vytížení dědí z `WorkloadException`, výjimky týkající se databáze z `DatabaseException`. Dědění zajišťuje jednodušší odchytávání v uživatelském rozhraní.

6.2.3 Extension

Extension metody, jsou speciální metody, kterými lze rozšířit funkčnost stávajících typů bez nutnosti dědění nebo změny původního typu. Implementována je jediná metoda `ReplaceFirst`, která v řetězci nahradí první výskyt hledaného podřetězce. Metoda je využita při zpracování XML souboru s vytížením, kdy nahrazujeme `?` hodnotami.

6.2.4 Factory

Jeden z nejpoužívanějších návrhových vzorů, sloužící pro tvorbu instancí objektů. Výhodou vzoru je, že při vytvoření instance ji můžeme ještě dodatečně inicializovat podle vstupních parametrů tovární metody. Druhou výhodou je, že nemusíme znát konkrétní typ objektu, stačí nám znát rozhraní, které objekty implementují. Továrna pak sama podle vstupních parametrů rozhodne, jakou konkrétní instanci vytvoří. V aplikaci jsou implementovány následující továrny:

6.2.4.1 Factory.Database.ConnectionDataFactory

Továrna na data pro připojení k databázi typu `ConnectionData` (viz kapitola 6.2.6.1), která poskytuje metody pro vytvoření objektu pro Windows Authentication a SQL Server Login. Dále obsahuje metody, které umí `ConnectionData` vytvořit ze settings WPF aplikace (viz kapitola 7.1.1).

6.2.4.2 Factory.Database.ServerFactory

Obsahuje jedinou metodu, která tvoří instanci typu `Server` z knihovny SMO a podle vstupního parametremu typu `ConnectionData` jej navíc inicializuje.

6.2.4.3 Factory.Workload.SqlQueryFactory

Továrna vytváří objekty `SqlQuery` (viz kapitola 6.2.6.3), buď z textu dotazu nebo z textu dotazu z XML vytížení a hodnot, které se dosadí za `?`. V případě, že vstup nesplňuje podmínky pro zpracování, vyhodí továrna výjimku s chybovou zprávou.

6.2.4.4 Factory.Workload.WorkloadLoaderFactory

Podle přípony souboru, která je vstupem tovární metody, vytvoří instanci implementující rozhraní `IWorkloadLoader` (viz kapitola 6.2.7.4).

6.2.4.5 Factory.Workload.WorkloadQueryTypeFactory

Vstupem je objekt typu `SqlQuery`, jehož text se v továrně zparsuje a podle typu dotazu se vytvoří konkrétní instance třídy implementující rozhraní `IQueryType` (viz kapitola 6.2.7.5). V případě, že budeme chtít podporu zpracovávaných dotazů rozšířit, stačí typ identifikovat v této továrně a vytvořit pro něj třídu implementující `IQueryType`.

6.2.5 Helper

Jedná se o pomocné třídy, které mají jen statické metody.

6.2.5.1 Helper.DataSetHelper

Obsahuje statické metody k práci s `DataSety`, které se v nástroji hojně používají. Dokáže pomocí LINQu provést sjednocení, rozdíl a průnik dvou `DataSetů`, které mají stejný počet sloupců a vrátit výsledný `DataSet`.

6.2.6 Model

Modely jsou jednoduché objekty, které mají za úkol přechovávat informace. Většinou jsou v nich definované jen vlastnosti (properties) a objekty nemají žádnou možnost s těmito daty manipulovat a měnit je.

6.2.6.1 Model.Database.ConnectionData

Objekt uchovává informace pro připojení k databázi — host, název databáze, typ připojení, případně uživatelské jméno a heslo.

6.2.6.2 Model.Workload.ExecutionPlan

Reprezentuje plán vykonání SQL dotazu a obsahuje počet a názvy operací, která jsou dále využívána pro porovnání plánů ze zdrojové a cílové databáze.

6.2.6.3 Model.Workload.SqlQuery

Dekorátor objektu `SqlCommand`, který přidává další vlastnosti, určující např. jestli je dotaz validní, jestli byl zpracován při subsettingu, jaký počet výsledků vrací na zdrojové i cílové databázi a jak dlouho trval jeho subsetting. Zajímavostí je uchovávání chybových hlášek. K tomu slouží

vlastnost typu `Dictionary`, kde klíč je hodnota enumu `SqlQueryErrorType` a určuje typ chybové zprávy. V hodnotě slovníku je pak uložena samotná hláška, kterou je právě podle typu možné vypsat na konkrétním místě v uživatelském rozhraní. Při vytvoření instance se dotaz ihned validuje pomocí `SqlQueryValidatoru` (viz kapitola 6.2.7.9)

6.2.6.4 Model.Workload.Table

Objekt nereprezentuje databázovou tabulku, uchovává o ni pouze určité informace. Třída je využita při testování po provedení subsettingu a obsahuje vlastnosti, které nám říkají kolik záznamů tabulka obsahuje ve zdrojové a cílové databázi a procentuální podíl přenesených záznamů.

6.2.7 Service

Služby reprezentují nějaké užitečné objekty, které by měla být zodpovědné jen za jednu konkrétní úlohu. Kdybychom měli v aplikaci např. službu `Mailer`, starala by se jen o odesílaní e-mailů. Rozdělením kódu do služeb získáme organizovanou architekturu, kdy přesně víme k čemu jednotlivé třídy slouží.

6.2.7.1 Service.Database.DatabaseCloner

Služba pro přenos schématu zdrojové databáze do cílové databáze.

V průběhu vývoje nastalo v této části množství problémů. Nejprve byl přenos schématu proveden třídou `Transfer` z knihovny SMO, které je určena právě pro klonování databáze. Použití této třídy se bohužel ukázalo jako časově neefektivní řešení u větších databází. Další možností byla třída `Script` také z knihovny SMO, ale její možnosti nebyly dostatečné. Aktuální řešení je popsáno níže.

Při procesu zálohování je využita třída `Backup` z knihovny SMO, pomocí které zálohu dočasně uložíme do aplikační dat nástroje. Ta se poté obnoví do nové databáze pomocí SQL dotazu, který je možné vidět ve výpisu 3. Cesty k databázovým souborům databáze uchovává jsme schopni zjistit z objektu databázového připojení a pomocí knihovny SMO. Další možný způsob je obnovení pomocí třídy `Restore`, při kterém však nastaly potíže, kdy nešly přenést potřebné databázové soubory ani při implementaci podle oficiální dokumentace. Po obnovení se smažou cizí klíče a na všech tabulkách provede `TRUNCATE`, tím databázi vyprázdníme. Poté se původní cizí klíče obnoví. Posledním krokem je smazání dočasného souboru zálohy z aplikačních dat.

```
USE master;
RESTORE DATABASE [SOURCE_DATABASE_NAME]
FROM DISK = 'PATH_TO_BACKUP'
WITH RECOVERY,
MOVE 'PATH_TO_SOURCE_DATABASE_MDF_FILE'
TO 'REQUIRED_PATH_TO_DESTINATION_DATABASE_MDF_FILE',
MOVE 'PATH_TO_SOURCE_DATABASE_LDF_FILE'
```

```
TO 'REQUIRED_PATH_TO_DESTINATION_DATABASE_LDF_FILE',
REPLACE
```

Výpis 3: Dotaz pro obnovení zálohy databáze

6.2.7.2 Service.Database.DatabaseConnection

Reprezentuje databázové připojení. K práci s databází jsou implementovány metody jako `Connect`, `Disconnect`, `CreateDatabase` a další. Níže jsou popsány zajímavé funkce z této třídy:

- `IgnoreDuplicateKeys` —- Zapne nebo vypne `IGNORE_DUP_KEY` na všech tabulkách v databázi. `IGNORE_DUP_KEY` určuje, jak se databáze chová při pokusu vložení záznamu s primárním klíčem, který již existuje. Implicitně je možnost nastavena na OFF a při pokusu vyhodí databáze výjimku a provede rollback celého dotazu. Při nastavení ON vypíše chybovou hlášku a vkládá jen neduplicitní záznamy, duplicitní ignoruje. Mód funkce se určuje parametrem typu `IgnoreDuplicateKeysMode`.
- `BulkInsert` — Prvním způsobem vkládání dat byl *batch insert*, který umožňuje jedním příkazem `INSERT` vložit do databáze více záznamů. V průběhu implementace se projevilo omezení, které umožňuje vložit pouze 1000 záznamů najednout. Prvotní nápad byl rozložit dotaz na více příkazů `INSERT`, každý s 1000 hodnotami. Poté jsem ale našel třídu `SqlBulkInsert`, které stačí předat `DataSet` a název tabulky a třída se o vše postará za nás. Metoda vrací počet vložených záznamů.
- `Count` — Při testování na větších databázích se vyskytl problém při subsettingu tabulek, které měly velký počet záznamů. Zdrojem problému byla operace `count(*)` nad takovou tabulkou. Funkce `count` s hvězdičkou nebere v potaz indexy a pro vrácení výsledku musí udělat *table scan*, což je velice neefektivní. Řešení je využít systémový katalog, pomocí kterého lze vrátit počet záznamů jak pro všechny tabulky najednou, tak i pro specifickou. Místo funkce `count(*)` je v aplikaci využíván dotaz 4.

```
SELECT ddps.row_count
FROM sys.indexes AS i
INNER JOIN sys.objects AS o ON i.OBJECT_ID = o.OBJECT_ID
INNER JOIN sys.dm_db_partition_stats AS ddps ON i.OBJECT_ID = ddps.
   OBJECT_ID AND i.index_id = ddps.index_id
WHERE i.index_id < 2 AND o.is_ms_shipped = 0
AND o.name = [table_name] -- nazev tabulky
ORDER BY o.NAME
```

Výpis 4: Náhrada funkce count(*)

Při práci s databází se využívá tzv. *connection pool*, což je cache databázových připojení. V *connection poolu* jsou udržované aktivní připojení, aby mohly být při požadavku na databázi znovupoužité a zabránilo se neustálému otevírání nových připojení, protože je to časově náročná operace. V aplikaci se proto před každým požadavkem na databázi volá metoda `Connect` a po požadavku `Disconnect`.

6.2.7.3 Service.FileAccess.CommonApplicationData

Slouží pro ulehčení zápisu a čtení souborů. Při implementaci nastal problém, kdy nebylo možné ukládat soubory na disk s konzistentními výsledky napříč vývojovými prostředími. Hlavním důvodem je, že každý uživatelský účet má jiná práva v operačním systému a tento účet nemusí mít práva k zápisu do uvedené složky, kde chceme data ukládat.

Praktickým příkladem je případ, kdy byla záloha databáze ukládána do `C:\Windows\Temp`. Potíže nastaly až při testování aplikace na jiném stroji, na kterém do tohoto umístění nešlo zapisovat. Jednoduchým řešením by bylo spustit aplikaci pod administrátorským účtem, ale tomuto přístupu jsem se chtěl vyhnout, protože v aplikaci pro to neexistuje žádný jiný důvod než tento. Mnohem lepším řešením je vytvoření složky pro aplikační data a přidělení potřebných práv. Samotná práce s třídou je už pak velmi jednoduchá záležitost. [7]

6.2.7.4 Jmenný prostor Service.WorkloadLoader

Celý jmenný prostor obsahuje třídy pro načítání dotazů z vytížení. Při návrhu opět hrála důležitou roli případná rozšiřitelnost a proto je zde využitý návrhový vzor *Strategie*, který nám umožňuje použití více algoritmů pro jednu úlohu. Základem je rozhraní `IWorkloadLoader`, které nařizuji implementaci pouze jedné metody, která vrací `List` objektů `SqlQuery`. Konkrétní třídy implementující toto rozhraní jsou `SqlWorkloadLoader` a `XmlWorkloadLoader` a zajišťují načtení dotazů ze souboru. Pro přidání dalšího typu vytížení stačí vytvořit novou třídu implementující `IWorkloadLoader` a v továrně `WorkloadLoaderFactory` (viz kapitola 6.2.4.4) přidat případ kdy se má vytvořit její instance.

6.2.7.5 Jmenný prostor Service.WorkloadProcessor.QueryType

Každá třída v tomto jmenném prostoru zajišťuje správný subsetting pouze danému typu dotazu. Momentálně se v aplikaci rozlišují dva typy dotazů, které se dále dělí na několik podtypů. Rozdělení je převzaté z dokumentace a praktických příkladů na stránkách *General SQL Parseru* [8].

1. *Ordinary* — Všechny dotazy, které nepatří do skupiny *combined*. Patří zde i dotazy prováděné nad více tabulkami (obsahující spojení klíčovým slovem `JOIN`) a vnořené dotazy.

2. *Combined* — Dotazy, které obsahují množinový operátor tzn. klíčové slovo `UNION`, `UNION ALL`, `EXCEPT` nebo `INTERSECT`. Existují i další, ale jen tyto čtyři jsou podporované SQL Serverem.

Pro každý dotaz musíme přenést všechna data, která jsou potřebná k jeho vykonání. Nestačí nám tedy pouze data, která získáme vykonáním dotazu, ale musíme zkopírovat i záznamy pro sloupce, které jsou obsažené např. v klauzuli `WHERE` nebo `ORDER BY`. Také v případě, že je dotaz prováděn nad více tabulkami, je nutné přenést data pro všechny tyto tabulky. Tento proces je vysvětlen algoritmem 2.

Vstup : Databáze zdroj a subset, dotaz dotaz
Výstup: Provedený subsetting pro daný dotaz z vytížení

```
1 foreach tabulka T obsažená v dotaz do
2 |  prefix ← jméno tabulky T
3 |  if T má alias then
4 |  |  prefix ← alias tabulky T
5 |  end
6 |  atomickyDotaz ← Nahraď vybírané sloupce za všechny sloupce tabulky T s prefixem
  |   prefix
7 |  data ← výsledek provedeného dotazu atomickyDotaz na zdroj
8 |  Vlož data do subset
9 end
```

Algoritmus 2: Subsetting dotazu nad více tabulkami

```
SELECT *
FROM Zakaznik z
JOIN Objednavka o ON z.ID_Zakaznik = o.ID_Zakaznik
WHERE ID_Zakaznik = 5;
```

Výpis 5: Příklad SQL dotazu s klíčovým slovem `JOIN`

Dotaz z příkladu 5 se při zpracování rozdělí na dva dotazy, které je možno vidět ve výpisu 6.

```
SELECT z.*
FROM Zakaznik z
JOIN Objednavka o ON z.ID_Zakaznik = o.ID_Zakaznik
WHERE ID_Zakaznik = 5;

SELECT o.*
FROM Zakaznik z
JOIN Objednavka o ON z.ID_Zakaznik = o.ID_Zakaznik
WHERE ID_Zakaznik = 5;
```

Výpis 6: Rozdělení dotazu z předchozího příkladu

V případě dotazů s množinovými operátory se nejprve prováděl subsetting pro každou část zvlášť, tzn. každý samostatný dotaz se zpracoval algoritmem 2. Toto řešení v některých případech způsobovalo nekonzistenci dat, protože se z nějakého důvodu úspěšně provedl subsetting jen pro první část dotazu. Proto je nyní v aplikaci implementováno lepší řešení, kdy se stále provede každá část dotazu zvlášť, ale poté se nad výsledky provede daná množinová operace pomocí třídy `DataSetHelper`. Důvodem tohoto způsobu je, že potřebujeme zjistit pro každý samostatný dotaz, nad kterými tabulkami je vykonáván a přenést data pro každou obsaženou tabulku.

V případě, že se ve vytížení objeví dotaz, pro který se subsetting neprovede správně, stačí vytvořit novou třídu implementující rozhraní `IQueryType` a v továrně `WorkloadQueryTypeFactory` určit, ve kterém případě se má instance této nové třídy vytvořit. Správnou implementací metody `Subset` zajistíme korektní subsetting pro tento typ dotazu.

6.2.7.6 Service.WorkloadProcessor.WorkloadProcesser

`WorkloadProcessor` je hlavní službou celého nástroje a stará se o kompletní provedení database subsettingu. Při inicializaci nového objektu této třídy se předávají informace o připojení ke zdrojové a cílové databázi v podobě `ConnectionData` objektů a informace o tom, zda chceme zachovat referenční integritu.

Třída nabízí jednoduché rozhraní, které se skládá jen ze 3 metod — `Init`, `ProcessOne` a `Close`. Metody jsou podrobněji popsány v textu níže.

1. `Init` — Slouží pro dodatečnou inicializaci objektu po jeho vytvoření. Po zavolání vytvoří obě databázové připojení a naklonuje schéma do cílové databáze s využitím třídy `DatabaseCloner`. Před jakoukoliv prací s touto třídou je nutno zavolat tuto metodu. Klonování je poměrně časově náročný proces a pro větší kontrolu se tyto akce neprovádí v konstruktoru. Také se zde zapíná nastavení, při kterém SQL Server ignoruje pokusy o vložení duplicitního klíče (viz kapitola 6.2.7.2).

2. `ProcessOne` — Metoda pro zpracování jednoho validního dotazu z vytížení typu `SqlQuery`. Vše co metoda dělá, je že předá objekt dotazu do továrny `WorkloadQueryTypeFactory` a na navráceném `IQueryType` objektu zavolá metodu `Subset`. Pokud se z jakéhokoliv důvodu nepovede dotaz zpracovat, metoda vyhodí výjimku `WorkloadException` s chybovou hláškou.

3. `Close` — V případě, že se má zachovat referenční integrita, dokopíruje tato metoda závislá data. V poslední řadě metoda dále zruší nastavení, díky kterému se ignorují pokusy o vložení duplicitního klíče.

Algoritmem 3 je popsán přenos závislých dat, tak aby zůstala zachována referenční integrita. Při tomto procesu musíme vzít v našem případě v potaz, že pokud je primární klíč typu `string` nebo `UUID` je nutné ho dát do jednoduchých uvozovek, kvůli tvorbě SQL dotazu. V algoritmu je

zahrnuta filtrace, díky které docílíme toho, že se vkládají pouze záznamy, které v cílové databázi ještě zaručeně nejsou. Tímto se šetří výpočetní čas, který by spotřebovaly zbytečné pokusy o vložení duplicitních záznamů.

Vstup : Databáze zdroj a subset
Výstup: Přenos dat potřebných pro zachování referenční integrity

```
1  frontaTabulek ← všechny tabulky cílové databáze
2  while frontaTabulek není prázdná do
3  |  T ← první tabulka z frontaTabulek
4  |  Odeber tabulku T z frontaTabulek
5  |  foreach ciziKlic FK v T do
6  |  |  hodnotyKlice ← zjisti a ulož hodnoty sloupce klíče FK v subset
7  |  |  idJizVlozenychZaznamu ← zjisti a ulož id již vložených záznamů v tabulce
   |  |   referencované klíčem FK v subset
8  |  |  idNevlozenychZaznamu ← prázdná kolekce
9  |  |  foreach hodnota ID v hodnotyKlice do
10 |  |  |  if ID není null a ID není v idJizVlozenychZaznamu then
11 |  |  |  |  if ID je typu string nebo UUID then
12 |  |  |  |  |  ID ← ID v jednoduchých uvozovkách
13 |  |  |  |  end
14 |  |  |  |  idNevlozenychZaznamu ← ID
15 |  |  |  end
16 |  |  end
17 |  |  if idNevlozenychZaznamu není prázdné then
18 |  |  |  data ← vyber záznamy s id z idNevlozenychZaznamu na databázi zdroj z
   |  |  |   tabulky, kterou referencuje klíč FK
19 |  |  |  Vlož data do databáze subset
20 |  |  |  if referencovaná tabulka má cizí klíč a není v frontaTabulek then
21 |  |  |  |  frontaTabulek ← referencovaná tabulka
22 |  |  |  end
23 |  |  end
24 |  end
25 end
```

Algoritmus 3: Přenos závislých dat

6.2.7.7 Service.WorkloadTester.TableTester

Jednoduchá služba, která při testování zjistí informace o jednotlivých tabulkách databáze. Jediná metoda třídy vrátí informace o všech tabulkách v podobě kolekce objektů `Table` (viz kapitola 6.2.6.4).

6.2.7.8 Service.WorkloadTester.WorkloadTester

Slouží ke kontrole správnosti provedeného subsettingu. Pro všechny zpracované dotazy zjistí, jestli vrací stejný počet výsledků na zdrojové i cílové databázi a jestli mají stejný plán vykonání

(viz kapitola 4.2).

Porovnání plánu vykonání je provedeno jeho analýzou v XML formátu. Porovnává se počet operací a jejich typ. Posledním testem, který se provede je spuštění `EXEC sp_msforeachtable "ALTER TABLE ? WITH CHECK CHECK CONSTRAINT ALL";` na podmnožině databáze. Pokud databázový server nevrátí chybu, znamená to, že se podařilo aktivovat omezení primárních a cizích klíčů a databáze nemá porušenou referenční integritu.

6.2.7.9 Service.Validator.SqlQueryValidator

Třída pro validaci SQL dotazů z vytížení před spuštěním samotného subsettingu. Samozřejmě by šlo zkusit provést subsetting pro všechny dotazy a zachytávat výjimky z SQL Serveru a ty předat uživateli, ale ten by se dozvěděl co je špatně až poté. Velkou výhodou validace před subsettingem je, že uživatel si vytížení může upravit, tak aby bylo zpracovatelné a neztrácí tedy čas. Veškerá validace je prováděna pomocí výjimek, které se poté odchytávají a zpracovávají v jiných částech aplikace. Veškeré dotazy musí splňovat dvě hlavní podmínky: mít bezchybnou syntaxi a být typu `SELECT`.

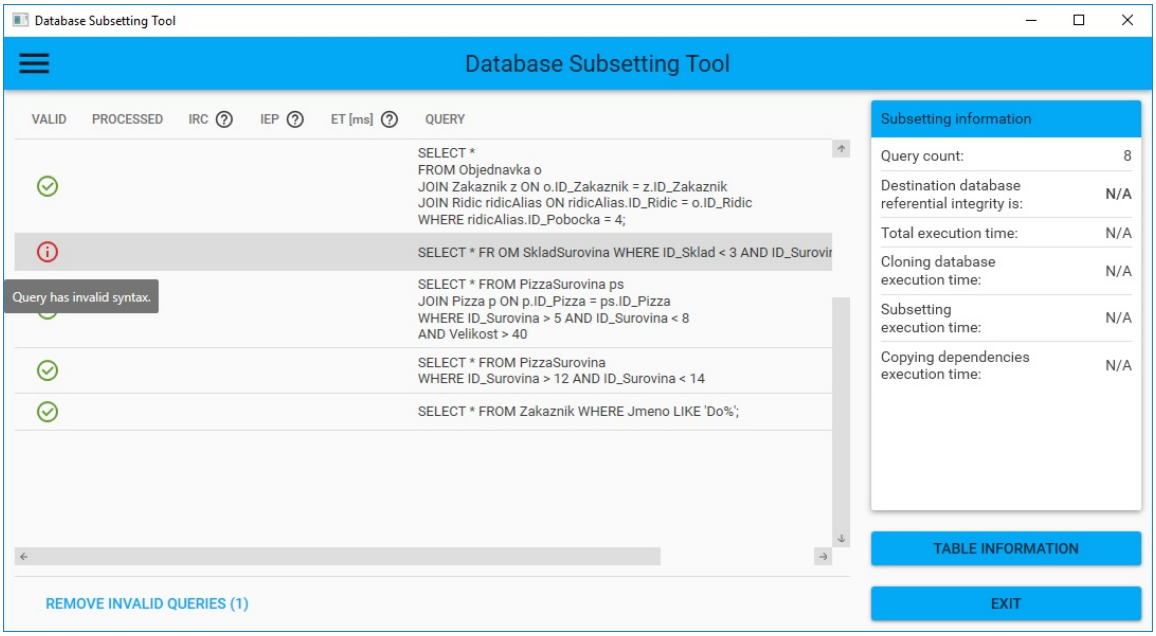
Pro kombinované dotazy (viz kapitola. 6.2.7.5) existují specifičtější podmínky. Aplikace nepodporuje víceúrovňové kombinované dotazy, to znamená, že dotaz může obsahovat maximálně jeden množinový operátor. Dále musí obě části, tzn. oba samostatné dotazy oddělené množinovým operátorem, vracet stejné sloupce a být provedeny nad stejnými tabulkami. S tím se váže zakázaní použití hvězdiček v těchto dotazech. SQL Server si s tím sice dokáže poradit, ale v naší aplikaci není možné takovýto dotaz analyzovat předem. V poslední řadě musí mít všechny tabulky obsažené v dotazu specifikovaný alias a každý atribut musí být prefixován jedním z těchto aliasů.

7 Aplikace

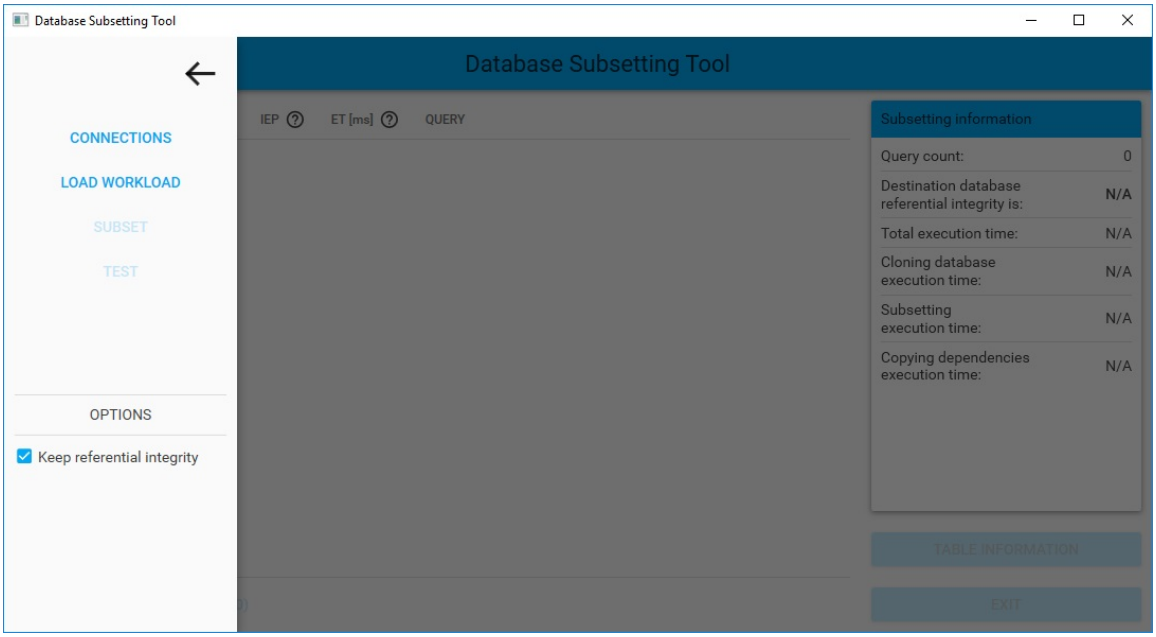
Po spuštění se otevře hlavní okno aplikace (viz obrázek 2). Hlavní funkce nástroje jsou umístěny v postranním vysouvacím menu, které se zobrazí po kliku na ikonu v levé horní části (viz obrázek 3).

Většinu prostoru zabírá `DataGrid`, které se naplní po načtení souboru s vytížením a při spuštění funkcí z menu se doplňuje dalšími hodnotami. Ikony indikují správnost nebo nesprávnost provedené akce a při najetí myší zobrazí chybovou hlášku. Tlačítkem pod `DataGridem` lze vyfiltrovat nevalidní dotazy.

Napravo nalezneme kartu s informacemi o subsettingu, např. počet dotazů, stav referenční integrity po subsettingu a časy jednotlivých kroků, se kterými se dále pracuje v kapitole 8.



Obrázek 2: Hlavní okno aplikace



Obrázek 3: Menu v hlavním okně

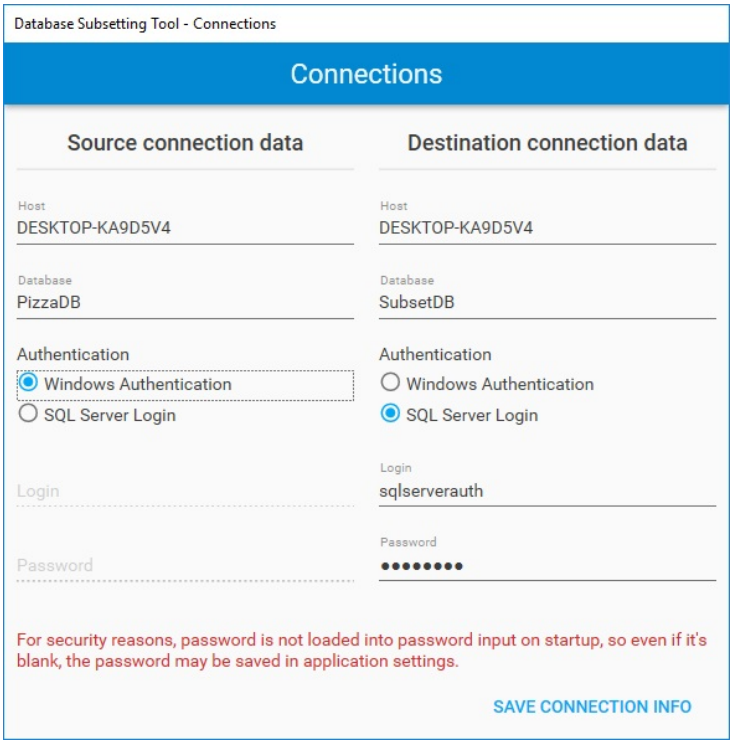
7.1 Akce v postranním menu

7.1.1 Připojení

Klikem na tlačítko *Connections* se otevře okno pro připojení k databázím (viz obrázek 4). Lze se připojit buď přes *Windows Authentication* nebo *SQL Server Authentication*. Cílová databáze se tvoří v průběhu subsettingu, proto je nutné zadat název databáze, která ještě neexistuje. V případě, že již existuje, zobrazí se okno s chybovou hláškou.

Informace o připojení z toho okna se ukládají do nastavení aplikace (`Settings.settings`) pod předem definovanými klíči. Celý objekt `Settings` je pak možné předat tovární metodě třídy `ConnectionDataFactory`, která podle těchto klíčů dokáže vytvořit objekty `ConnectionData`.

Z důvodu bezpečnosti není možné ve WPF bindovat komponentu `PasswordBox`, proto je v aplikaci poznámka, že se heslo nemusí zobrazovat i přesto, že je vyplněné.



Obrázek 4: Okno pro připojení k databázím

7.1.2 Načtení dotazů z vytížení

K načtení slouží tlačítko *Load Workload*. Po kliku se zobrazí klasický dialog pro vybrání souboru omezen na typy `*.sql` a `*.xml`. K načítání dotazů jsou použity třídy `SqlWorkloadLoader` a `XmlWorkloadLoader` (viz kapitola 6.2.7.4).

Po výběru se dotazy ze souboru načtou do `DataGridu` a zvalidují. Pokud je dotaz chybný (viz kapitola 6.2.7.9), lze to vidět ve sloupci `Valid` a při najetí myší na ikonu se zobrazí konkrétní chybová hláška (viz obrázek 2).

Při zobrazení v `DataGridu` je zachováno formátování textu ze souboru vytížení.

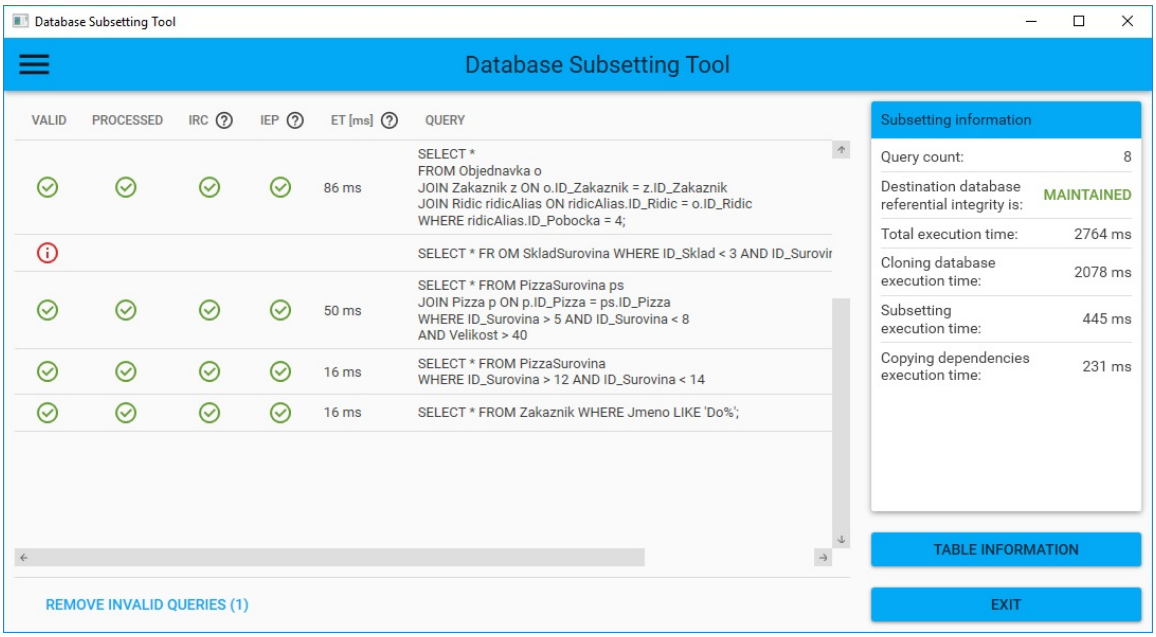
7.1.3 Provedení database subsettingu

Tlačítko *Subset* zahajuje database subsetting. Tato akce využívá zejména třídy `DatabaseCloner` (viz kapitola 6.2.7.1) a `WorkloadProcesser` (viz kapitola 6.2.7.6). Zavoláním metody `Init` se inicializují databázová připojení a naklonuje databáze. Poté se iteruje nad kolekcí dotazů z vytížení a postupně se zpracovávají. Pokud dojde k chybě, je odchycena pomocí `try-catch` bloku, ve kterém se nastaví atribut `Processed` na `false` a do chybových hlášek se přidá text z výjimky, kterou lze zobrazit najetím na ikonu ve stejnojmenném sloupci. Také je u každého dotazu ve sloupci `ET` vidět čas, který jeho subsetting zabral.

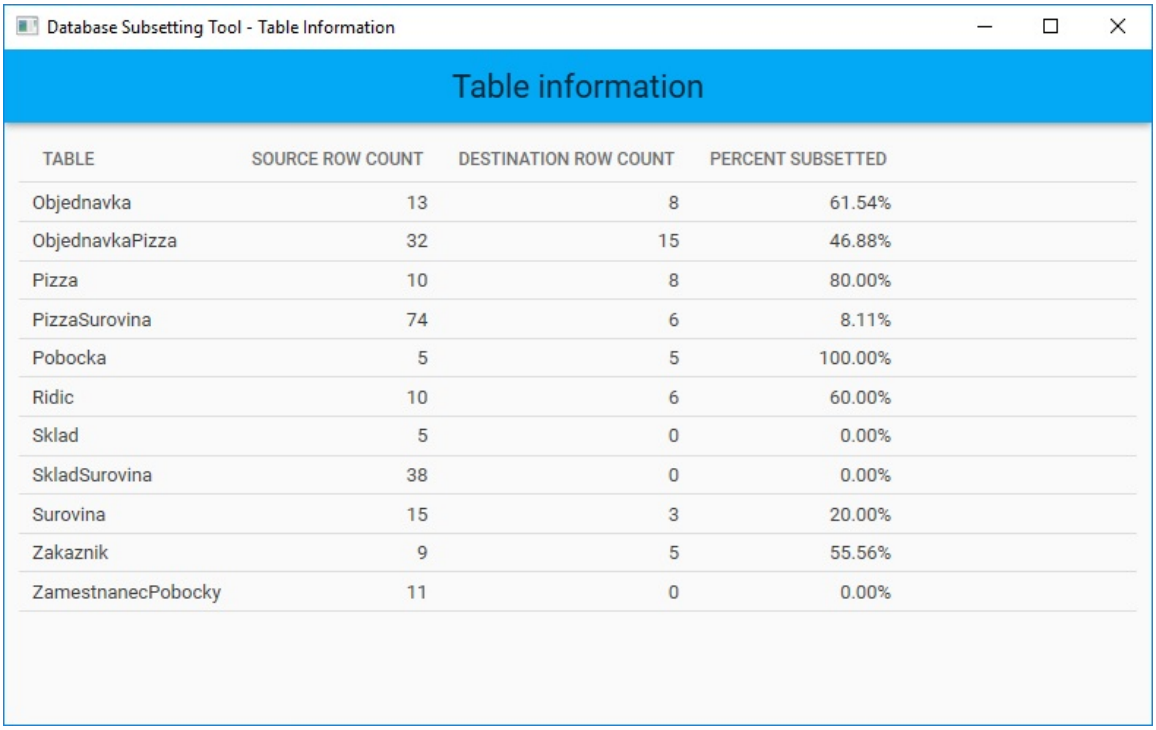
7.1.4 Testování po subsettingu

Posledním krokem je automatický test provedeného database subsettingu. Ten spustíme kliknutím na tlačítko *Test*. Akce využívá třídy `WorkloadTester` (viz kapitola 6.2.7.8) a `TableTester` (viz kapitola 6.2.7.7). Po otestování se v `DataGridu` doplní sloupce `IRC` (identický počet výsledků na zdrojový databázi i podmnožině) a `IEP` (identický plán vykonání na obou databázích). Výsledek můžeme vidět na obrázku 5.

Po kliku na tlačítko *Table information* se zobrazí okno s informacemi o jednotlivých tabulkách, konkrétně počet vracených záznamů na zdrojové databázi i podmnožině a procento přenesených dat. Toto okno je na obrázku 6.



Obrázek 5: Hlavní okno po testu subsettingu



Obrázek 6: Okno s informacemi o subsettingu jednotlivých tabulek

7.1.5 Nastavení

Poslední položkou v menu je nastavení. Před spuštěním subsettingu je možné vypnout nebo zapnout zachování referenční integrity. V případě, že nastavení zapneme, nakopírují se všechna závislá referencovaná data (viz kapitola 6.2.7.6). V opačném případě se přenesou jen data, které dotazy reálně vrací, což může ušetřit čas, pokud potřebujeme opravdu jen tyto data.

8 Testování

8.1 Časy subsetování

Pro každou databázi a dané vytížení z tabulek 5, 6 a 7 byly provedeny nejméně tři testy a ty následně zprůměrovány.

Z tabulek vyplývá, že největší časovou náročnost má subsetting složitějších dotazů, např. vnořené nebo prováděné nad více tabulkami. To je dáno tím, že samotné vykonání takového dotazu v SQL Serveru trvá déle, protože je náročnější ho zpracovat.

Podstatnou část času zabere klonování databáze. Toto je také očekávané, musí se provést záloha, obnovení a zkopírovaní fyzických souborů do jiného umístění na disku.

Detailnější výsledky subsettingu v podobě Excel tabulek lze nalézt na přiloženém médiu ve složce *appendix*. Ve stejné složce jsou také umístěny SQL a XML soubory s vytížením.

Nástroj byl testován na notebooku Lenovo G580, Intel Core i7-3520M 2.90GHz, RAM 8GB, 256GB SSD, OS Windows 10 a byla použita aplikace sestavená jako *Release*, tzn. produkční verze. Vždy byla zvolena možnost zachování referenční integrity.

Tabulka 5: Čas subsettingu na databázi PizzaDB

	Kopírování struktury databáze	**Subsetting**	**Dokopírování závislých dat**	**Celkový čas celého procesu**
Průměr [ms]	2030	1333	725	292
Průměr [s]	2,03	1,33	0,73	0,29

Tabulka 6: Čas subsettingu na databázi Aukce

	Kopírování struktury databáze	**Subsetting**	**Dokopírování závislých dat**	**Celkový čas celého procesu**
Průměr [ms]	2918	2075	165	5181
Průměr [s]	2.92	2,08	0,17	5,18

Tabulka 7: Čas subsettingu na databázi AdventureWorks

	Kopírování struktury databáze	**Subsetting**	**Dokopírování závislých dat**	**Celkový čas celého procesu**
Průměr [ms]	3570	507	603	4696
Průměr [s]	3,57	0,51	0,6	4,7

8.2 Výsledky subsetování

Na tabulkách 8, 9 a 10 lze vidět výsledky po provedeném subsettingu.

Tabulka 8: Výsledek subsetování na databázi PizzaDB)

Zdroj celkem [**záznamy**]	**Subset celkem** [**záznamy**]	**Velikost subsetu** [%]	**Soubor**	**Databáze**
222	65	29.27	pizzaDB.sql	PizzaDB

Tabulka 9: Výsledek subsetování na databázi Aukce

Zdroj celkem [**záznamy**]	**Subset celkem** [**záznamy**]	**Velikost subsetu** [%]	**Soubor**	**Databáze**
378 169	211	0.557	aukce.xml	Aukce

Tabulka 10: Výsledek subsetování na databázi AdventureWorks

Zdroj celkem [**záznamy**]	**Subset celkem** [**záznamy**]	**Velikost subsetu** [%]	**Soubor**	**Databáze**
281 914	1291	0.458	adventureWorks.sql	AdventureWorks

8.3 Identický plán vykonání

Jednou z podmínek práce je, že dotazy mají mít stejný plán vykonání na cílové databázi i podmnožině. Plán vykonání je ovlivněn počtem záznamů, indexy nebo optimalizací dotazu. Ani jedna z těchto možností není přípustná, protože bychom buď modifikovali samotný dotaz a už bychom neměli identické vytížení nebo bychom modifikovali databázi a to může ovlivnit testování nějaké aplikace mnohem více, např. chybějící index na sloupci, přes který se vyhledává, může aplikaci zpomalit.

Jediným způsobem jak lze ovlivnit plán vykonání bez těchto změn, je tzv. *plan forcing* (vynucení plánu). Plán vynutíme přidáním `OPTION (USE PLAN N'XML_EXECUTION_PLAN')`, kde `XML_EXECUTION_PLAN` je XML plán dotazu na původní databázi. Tento krok není trvalý, musí se provádět s každým dotazem. Neexistuje proto způsob, jak plán ovlivnit z naší aplikace. Záleží pouze na uživateli vytížení, zda si k dotazům přidá vynucení pomocí `OPTION`. [9]

9 Závěr

Hlavním cílem práce bylo vytvořit nástroj pro database subsetting s ohledem na vytížení. Podmínkou bylo, že dotazy z vytížení budou vracet stejný počet záznamů. Dotazy by také měly mít na obou databázích stejný plán vykonání.

Práce je rozdělena na 2 části, které jsou dále rozděleny na kapitoly. První část je teoretická a zabývá se vysvětlením toho, co database subsetting je, proč jej používat a problémy, které při něm mohou nastat. Také v této části nalezneme rešerši a porovnání existujících nástrojů.

V druhé části je popis samotné implementace nástroje. Na začátku této části jsou představeny použité technologie s podrobnějším popisem parserů. Detailně je zde popsána architektura a algoritmy složitějších funkcí programu.

Při implementování a testování jsem narazil na několik problémů, které byly zapříčiněny buď špatnou implementací nebo byly důsledkem limitů použité technologie. Všechny problémy, které nastaly jsou v práci popsány a součástí je také vysvětlení řešení těchto problémů. Nástroj podle mého názoru a testování splňuje všechny požadavky zadání.

V rámci dalšího vývoje by určitě na prvním místě byla podpora více DBMS, optimalizace subsettingu složitějších dotazů a programátorské testování ve formě unit-testů. Dále by aplikace mohla nabízet export výsledků do souboru kvůli jednoduššímu testování. V rámci estetiky by se jednalo o doladění uživatelského rozhraní.

Téma práce pro mě bylo zajímavé a naučil jsem se díky němu mnoho nových věcí v oblasti technologií spojených s databázemi a .NET frameworkem, zejména knihovnou WPF. Také jsem si prohloubil znalosti týkající se návrhového vzoru MVVM a data-bindingu. Jak MVVM, tak data-binding jsou v dnešní době velmi rozšířené techniky, takže to považuji za velký přínos.

Literatura

[1] SHARMAN, Pete. *Data Subsetting* [online]. [cit. 2018-04-05]. Dostupné z: https://petewhodidnottweet.com/2015/04/data-subsetting/

[2] A Net 2000 Ltd. *Database Subsetting: What You Need to Know* [online]. [cit. 2018-04-05]. Dostupné z: https://www.databee.com/DatabaseSubsetting_WhatYouNeedToKnow.pdf

[3] *About The ANTLR Parser Generator* [online]. [cit. 2018-04-05]. Dostupné z: http://www.antlr.org/about.html

[4] *Getting Started with ANTLR v4* [online]. [cit. 2018-04-05]. Dostupné z: https://github.com/antlr/antlr4/blob/master/doc/getting-started.md

[5] *T-SQL (Transact-SQL, MSSQL) grammar* [online]. [cit. 2018-04-05]. Dostupné z: https://github.com/antlr/grammars-v4/tree/master/tsql

[6] Wikipedia. *Parsing* [online]. [cit. 2018-04-05]. Dostupné z: https://en.wikipedia.org/wiki/Parsing

[7] *Allow write/modify access to CommonApplicationData* [online]. [cit.2018-04-05]. Dostupné z: https://www.codeproject.com/Tips/61987/Allow-write-modify-access-to-CommonApplicationData

[8] *Decoding SQL grammar — Select SQL Statement with UNION, INTERSECT, and EXCEPT set operators* [online]. [cit.2018-04-05]. Dostupné z: http://www.dpriver.com/blog/list-of-demos-illustrate-how-to-use-general-sql-parser/decoding-sql-grammar-select-sql-statement-with-union-intersect-and-except-set-operators/

[9] *Understanding Plan Forcing* [online]. [cit.2018-04-05]. Dostupné z: https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms186343(v=sql.105)

A Struktura přiloženého média

Tabulka 11: Obsah média

Adresář	Obsah
\bp	PDF soubor
\src	Projekt s aplikací ve Visual Studiu 2017
\appendix	Externí přílohy (detailnější výsledky testování, soubory s vytížením)